MRPOD Documentation

Release 1.0.0

Zhiyao Yin

Jul 20, 2021

Contents

CHAPTER 1

Contents

See also:

For more detailed discussions on MRPOD, its derivations and applications, please see the publication [MRPOD]. We also kindly ask you to reference this paper if you use `mrpod` for your publications.

1.1 Greetings

Welcome to the documentation of the Python module `mrpod` for performing Multiresolution Proper Orthogonal Decomposition (MRPOD) of multi-dimensional time series. Although `mrpod` was created to tackle problems in turbulent flows, this module is equally applicable to various data series to achieve frequency-filtering, classification of dynamics, identification of discontinuities, etc. The built-in wavelet sub-module can be easily applied to 1-D and 2-D dataset for wavelet decomposition and reconstruction.

1.1.1 Why `mrpod`

Data-driven, modal-decomposition techniques such as POD (also known as the Principle Component Analysis, PCA) and Dynamic Mode Decomposition (DMD) have been widely implemented to extract periodic, coherent structures in turbulent flows. In reacting flows such as confined turbulent flames (in gas turbines), however, we are often confronted with both periodic (e.g., hydrodynamic and acoustic instabilities) and non-periodic dynamics (e.g., flame lift-off, flashback, and bistability), which can coexist over a wide range of time scales and may even interact with each other.

In their most rudimentary forms, POD and DMD have proven insufficient at separating these dynamics while resolving their temporal behaviors (such as discontinuities) at the same time. On the other hand, by marrying the concept of wavelet-based Multiresolution Analysis (MRA) with standard modal decompositions, multiresolution DMD ([MRDMD]) and multi-scale POD ([mPOD]) have demonstrated robust capabilities at identifying unsteady dynamics and discontinuities in time series.

Based on a similar concept, multiresolution POD ([MRPOD]) has been developed by combining Maximum-Overlap Discrete Wavelet Transform (MODWT) with conventional snapshot POD. MRPOD has been successfully applied to

time series of velocity (vector) and scalar fields obtained by kHz-rate laser diagnostics (e.g., Particle Image Velocimetry or PIV, Planar Laser-induced Fluorescence or PLIF) in the so-called bistable turbulent swirl flame, a common phenomenon encountered in gas turbines.

1.1.2 Why MODWT

`mrpod` was developed to satisfy primarily the following two criteria:

- Dynamics with various frequencies can be identified and adequately isolated.
- Discontinuities in temporal behaviors can be properly resolved and align perfectly with the original data series for appropriate comparison.

Unlike the classical DWT, shift-invariant DWT such as MODWT is well-defined for arbitrary sample sizes and is not sensitive to the "break-in" point in the time series. Additionally, MODWT affords a more "square-looking" gain response and hence a higher spectral isolation especially for cases with dynamics densely packed in the frequency domain. Although MODWT sacrifices orthonormality, it can still carry out an exact analysis of variance as well as a perfect reconstruction of the time series. With the two aforementioned criteria in mind, MODWT has demonstrated overall better performance than DWT at a (manageable) cost of computational time.

1.1.3 Why not `pywt`

Instead of using the existing Python library `pywt` to carry out wavelet transform, a matrix-operation based routine was written from the ground up specifically for more efficient 1-D and 2-D wavelet decomposition/reconstruction of multi-dimensional data series stored in ndarrays. Although several commonly used wavelet filters are built into `mrpod`, the vast library of wavelet filters in `pywt` should be taken advantage of when constructing custom composite filters using the filter-cascading method in `mrpod`.

1.2 Wavelet Transform

Matrix-based computation of discrete wavelet transform: 1. Construct the wavelet filter

class mrpod.wavelet_transform.**CompositeFilter**(*g=None*, *wavelet='D(8)'*)

Generate composite filter Composite filter generated from a given wavelet for a specific decomposition level and all the scales involved.

Parameters

g [1d array, optional] Scaling (lowpass) filter for calculating the wavelet filter.

wavelet [str, optional] Name of the scaling filter (g). Default is 'D(8)'. Built-in options are: 'Haar', 'D(4)', 'D(8)', 'D(10)', 'D(12)', 'LA(8)', 'LA(12)', 'LA(16)'

- 'D(8)': Daubechies standard wavelet with 8 elements.
- 'LA(8)': Least-asymmetric variant of the Daubechies wavelet with 8 elements.

Other wavelets can be found in the following sources:

- *Wavelet Methods for Time Series Analysis* by Percival and Walden.
- Import from the `pywt` package using `pywt.Wavelet()`. Note that only half of the wavelet length is indicated in the name of the wavelets. E.g., the equivalent of 'D(8)' and 'LA(8)' in `pywt` are 'db4' and 'sym4'.

Notes

The generated filters need to be normalized by $2^{j/2}$ for maximum overlap wavelet transforms. This is taken into account in `WaveletTransform`.

filter_cascade(*J*)

Populate the collection of filters corresponding to all the j and n in a cascading scheme based on the specified target decomposition level J.

Parameters

J [int] Target decomposition level (or the maximum decomposition level). It should be larger than 1.

max_J(*N*)

Maximum decomposition level for the data series with the given wavelet filter. Although there is technically no upper limit for MODWT, it is still recommended to apply the same criterion as in the case of DWT.

Parameters

N [int] Length of the data series

Returns

max_dec [int] (Recommended) Maximum decomposition level for the data series with the given wavelet filter.

save_filterbank(*full_path_write*)

Save the computed filterbank to the specified local directory.

Parameters

full_path_write [obj] Full path to the file, recommended to use "./filterbank_[filtertype]_[j].pkl" to name the file.

sqd_gain_fcn(*j*, *n*, *fs=None*, *max_overlap=False*)

Squared gain function of a given wavelet filter at subscale n and decomposition level j.

Parameters

j [int] Decomposition level j

n [int] Index of the scale from a certain decomposition level j, $n = 0, 1 \ldots, 2^{j-1} - 1$.

fs [None, 1d array, optional] Frequencies to be used for caclulating the transfer function [0, 1/2).

max_overlap [False, bool, optional] If True, the output represent the squared gain from MODWT.

Returns

gain [1d array] Squared gain for the specified filter

fs [1d array] Standard frequency range

u_n(*n*)

Filter corresponds to the scale n of a level j, i.e. a selection of g or h.

Parameters

n [int] Index of the scale from a certain decomposition level j, $n = 0, 1 \ldots, 2^{j-1} - 1$.

Returns

u_n [1d array] Filter coefficients (either g or h).

class mrpod.wavelet_transform.**WaveletTransform**(*X*, *j=None*, *mode='max-overlap'*, *filterbank=None*, *full_path_filterbank=None*)

Basic class for discrete wavelet (packet) transform.

Input either data or correlation matrix.

Parameters

X [ndarray] Input data. Two shapes are admissible: - NxM with N being the sample size and M the total amount of physical coordinates (for 1d wavelet transform) - NxN with N being the sample size (for 2d wavelet transform)

j [int] Target decomposition level. A value is expected if 'standard' mode is used.

mode ['max-overlap', str. optional] Two modes of discrete wavelet transform to choose from: - 'max-overlap', perform maximum overlap DWT - 'standard', perform DWT.

filterbank [None, dict, optional] Precalculated filterbank via `CompositeFilter()`. It is also possible to input custom filters in the following format: {Node: Coefficients}. Node has to follow the naming convention '[j][0]'.

full_path_filterbank [None, obj, optional] Path to the filterbank pickle file created using `save_filterbank()` in `CompositeFilter()`.

detail_bundle_1d(*js*, *scales*)

Reconstruct the input data with MODWT at specified j levels and with desired scales, along the axis=0 dimension of the data (i.e., N in an NxM data).

Parameters

js [1d array of int] The correponding decomposition levels. Different j levels are admissible.

scales: 1d array of int All the scales included in the reconstruction. Discrete scales are admissible.

Returns

B_details [ndarray] Reconstructed detail bundle of input data with the dimension of NxM.

Notes

js and *scales* need to have the same length.

detail_bundle_2d(*js*, *scales*)

Reconstruct the input data (2d) with MODWPT at specified j and with desired scales.

Parameters

js [1d arrray of int] The correponding decomposition levels. Different j levels are admissible.

scales: 1d array of int All the scales included in the reconstruction. Discrete scales are admissible.

Returns

K_mat [ndarray] Reconstructed input data with the dimension of NxN.

T_mat: ndarray Transform matrix used to perform the transform.

filter_matrix(*j*, *n*)

Convert the composite filter u_j into its matrix form based on the sample size of N.

Parameters

j [int] The decomposition level.

n [int] The scale in level j.

index_W [int] Indices of the matrix form of the wavelet filter.

Returns

u_j_mat [ndarray] Composite filter u_j in its matrix form

index_W
Matrix conversion indices for data series and wavelet coefficient.

power_spectrum_1d(*j*, *scales*)
Calculate the approximated power spectrum of the input data based on `wavelet_coeff_1d`.

Parameters

j [int] The decomposition level.

scales: list, 1d array of int All the scales included in the decomposition.

Returns

power_spectrum [1d array] Power spectrum at specific scales of j. It has a size of len(scales).

power_spectrum_2d(*j*, *scales*)
Calculate the power spectrum of the input data based on `wavelet_coeff_2d`.

Parameters

j [int] The decomposition level.

scales: list, 1d array of int All the scales included in the decomposition.

Returns

power_spectrum [1d array] Power spectrum at specific scales of j. It has a size of len(scales).

wavelet_coeff_1d(*j*, *scales*)
Decompose the input data with MODWPT at specified j and with desired scales, along the axis=0 dimension of the data (i.e., N in an NxM data). It is possible to combine multiple (discrete) scales to achieve desired filtering effect.

Parameters

j [int] The decomposition level.

scales: list, 1d array of int All the scales included in the decomposition

Returns

W_j [ndarray] Wavelet coefficients from decomposing input data. It has a dimension of len(scales)xNxM

Notes

1. Not recommended to set a large number of scales if M is very large
2. Better to start with a single scale

wavelet_coeff_2d(*j*, *scales*)

Decompose the input data with MODWT at specified j and with desired scales, along both axes of the data dimension. Only data with a shape of NxN is admissible due to the precomputed Designed for handling cross-correlation matrices.

Parameters

j [int] The decomposition level.

scales [list, 1d array of int] All the scales included in the decomposition.

Returns

W_j [ndarray] Wavelet coefficients from decomposing input data. It has a dimension of len(scales)xNxN.

mrpod.wavelet_transform.**find_scale_index**(*level*, *x='0'*, *y='1'*)

Gray code order is used here to generate indices for the scales of each decomposition level in a wavelet packet transform. Either 'a' and 'd' (approximation and detail) or '0' and '1' are used. E.g., for a level=1 decomposition, the possible combination will be 'ad' and 'da'. For level=2, it will be 'aa', 'ad', 'da', 'dd'. The indices are also ordered by their corresponding frequency bandpasses.

Parameters

level [int] Dcomposition level for WPT.

x ['0', str, optional] First index in the gray code. 'a' is also generally used.

y ['1', str, optional] Second index in the gray code. 'd' is also generally used.

Returns

graycode_order [list] The list of indices ordered by their corresponding frequencies.

mrpod.wavelet_transform.**scale_to_frq**(*f_sample*, *j*)

Convert scales at a given j to their corresponding center frequencies.

Parameters

f_sample [float] Sampling rate of the data.

j [int] The decomposition level.

Returns

frq [1d array] Center frequencies of all the scales at a given level j

mrpod.wavelet_transform.**time_shift**(*w_j*, *L*, *j*, *scale*)

Time shift the wavelet coefficient so that it matches the features in the original signal temporally. Only works if the half length of the wavelet is even and if the wavelet is of the LA type (symlet).

Parameters

w_j [1d array] Wavelet coefficient.

L [int] Length of the wavelet used to calculate the wavelet coefficient.

j [int] Decomposition level.

scale [int] The specific scale (n) the wavelet coefficient corresponds to.

Returns

w_j [1d array] Wavelet coefficient corrected for its corresponding time shift

mrpod.wavelet_transform.**transfer_fcn**(*coeff_filter*, *freq_domain*)

Transfer function of a given wavelet filter.

Parameters

coeff_filter [1d array] Coefficients of the filter

freq_domain [1d array] Frequencies to be used for calculating the transfer function

Returns

T_fcn [1d array] Transfer function of the given filter

1.3 Modal Decomposition

mrpod.modal_decomposition.**mrpod_detail_bundle**(*data_array*, **args*, *num_of_modes=50*, *seg=10*, *subtract_avg=False*, *reflect=False*, *normalize_mode=True*, *full_path_write=None*, ***kwargs*)

Computes MRPOD modes, eigvals and proj coeffs from a dataset arranged in an ndarray of the shape of NxM0xM1x..., with N being the dimension that will be wavelet transformed.

Parameters

data_array [ndarray] data array arranged in a dimension of NxM0xM1x..., such as a time series (N) of multi- dimensional scalar/vector fields.

num_of_modes [50, int, optional] Number of modes to be computed (from the most energetic Mode 1). By default, all the valid modes are computed.

seg [10, int, optional] In case of insufficient computer RAM, the `data_array` can be segmented along the M dimension of the reshaped NxM and pieced together after the filtering.

subtract_avg [False, bool, optional] If True, the ensemble average of the `data_array` along N is subtracted from the dataset.

reflect [False, bool, optional] If True, the cross-correlation matrix as well as the dataset is padded symmetrically along the sample axis.

normalize_mode [True, bool, optional] If True, the magnitudes of the modes will be normalized by their corresponding eingenvalues and the sample size.

full_path_write [None, path obj, optional] If provided, the output is saved locally in the specified directory.

Returns

dict A dictionary containing the eigenvalues, projection coefficients, POD modes, and the cross-correlation matrix.

Other Parameters

****kwargs :** Keyword arguments from `mrpod_eigendecomp()` necessary to perform a wavelet transform.

mrpod.modal_decomposition.**mrpod_eigendecomp**(*corr_mat*, *js*, *scales*, *pod_fcn=None*, *reflect=False*, ***kwargs*)

Apply eigenvalue decomposition to cross-correlation matrices reconstructed in specific bandpasses using `WaveletTranform`.

Parameters

corr_mat [ndarray] Cross-correlation matrix with a dimension of NxN

js [1d arrray of int] The correponding decomposition levels. Different j levels are admissible for "max-overlap".

scales [1d array of int] All the scales included in the reconstruction. Discrete scales are admissible.

pod_fcn [function object] Function to solve the eigenvalue problem by taking in a pre-computed cross-correlation matrix. The output must conform to [eigvals, proj_coeffs]. If None, the built-in solver `pod_eigendecomp` is used.

reflect [False, bool, optional] If true, the corr_mat is padded symmetrically along both axes and is truncated after reconstruction and before eigenvalue decomposition. Such measure helps reducing the effect of uneven boundaries in the wavelet transform process.

reflected [False, bool, optional] If true, the supplied corr_mat has already been padded and is truncated before fed into the eigenvalue solver.

Returns

eigvals [1d array] Eigenvalues from the decomposition

proj_coeffs [ndarray] Projection coefficients (temporal modes) of the shape of NxN

K [ndarray] Reconstructed cross-correlation matrix within the designed bandpasses.

Other Parameters

****kwargs :** Keyword arguments from `WaveletTransform` necessary to perform a wavelet transform.

`mrpod.modal_decomposition.`**`ortho_check`**(*v1*, *v2*)

Check the orthogonality of two vectors.

`mrpod.modal_decomposition.`**`pod_eigendecomp`**(*corr_mat*, *tol=1e-14*)

Solving the eigenvalue problem of AX=Lambda.

Parameters

corr_mat [ndarray] Cross-correlation matrix with a dimension of NxN.

Returns

eigvals [1d array] Eigenvalues from the decomposition.

proj_coeffs: ndarray Projection coefficients (temporal modes) of the shape of NxN.

`mrpod.modal_decomposition.`**`pod_modes`**(*data_array*, *pod_fcn=None*, *eigvals=None*, *proj_coeffs=None*, *num_of_modes=None*, *normalize_mode=True*)

Calculate the POD modes based on the eigenvalue decomposition.

Parameters

data_array [ndarray] Data array arranged in a dimension of NxM0xM1x..., such as a time series (N) of multi- dimensional scalar/vector fields.

pod_fcn [function object] Function to solve the eigenvalue problem by taking in a pre-computed cross-correlation matrix. The output must conform to [eigvals, proj_coeffs]. If None, the built-in solver `pod_eigendecomp` is used.

eigvals [1d array] Eigenvalues from the decomposition.

proj_coeffs: ndarray Projection coefficients (temporal modes) of the shape of NxN.

num_of_modes [None, int, optional] Number of modes to be computed (from the most energetic Mode 1). By default, all the valid modes are computed.

normalize_mode [True, bool, optional] If True, the magnitudes of the modes will be normalized by their corresponding eingenvalues and the sample size.

Returns

dict A dictionary containing the eigenvalues, projection coefficients, POD modes, and the cross-correlation matrix.

1.4 Tutorials

A few tutorials are included here to get you started with wavelet-based MRA, POD as well as MRPOD using the module `mrpod`.

1.4.1 Pattern recognition with POD

For this tutorial, we will look at how to decompose the following time series of (synthesized) vector fields that contains a typical vortex pattern found in swirl-stabilized combustors using POD:

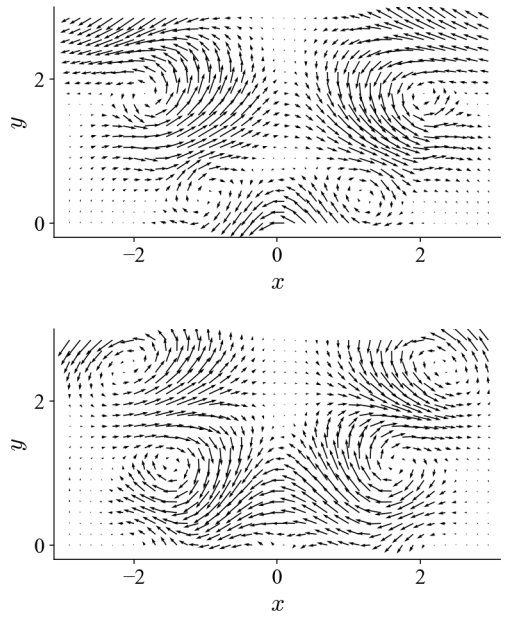
The goal is to try to recognize the flow pattern based on the POD results.

Synthesize dataset

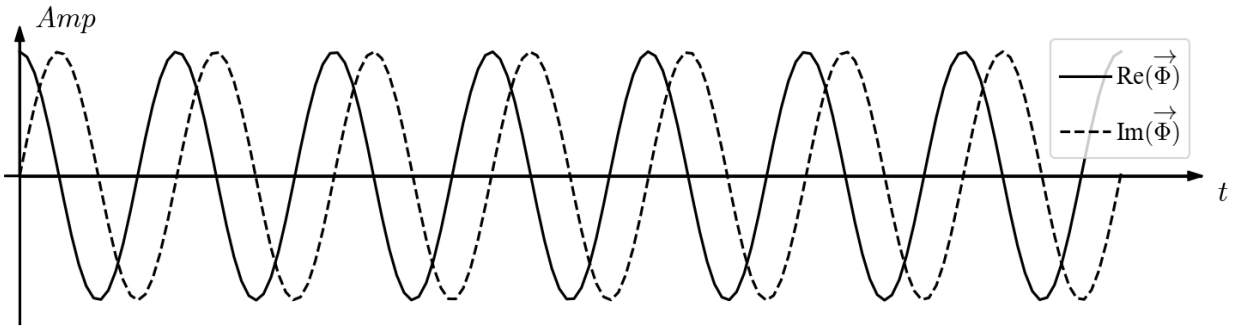
This vortex pattern is a signature of the 3-dimensional helical precessing vortex core in a 2D cut plane. This is usually seen in planar Particle Image Velocimetry measurements. To create these traveling vortices, two stationary modes are required as dictated by:

$$\boldsymbol{v}(x,t) = e^{-iwt}\boldsymbol{\Phi}(x) = \cos wt \cdot \Re(\boldsymbol{\Phi}) + \sin wt \cdot \Im(\boldsymbol{\Phi})$$

In this specific case $\Re(\boldsymbol{\Phi})$ and $\Im(\boldsymbol{\Phi})$ are constructed as:



respectively, with temporal behaviors set as:



Notice that the two stationary modes have a spatial shift of roughly a quarter of the wavelength and a temporal shift of 90 degrees (as implied in the equation above). Now that we have an artificial dataset, we can decompose it with the aim of identifying this vortex pattern. What kind of POD modes do we expect to extract out of it?

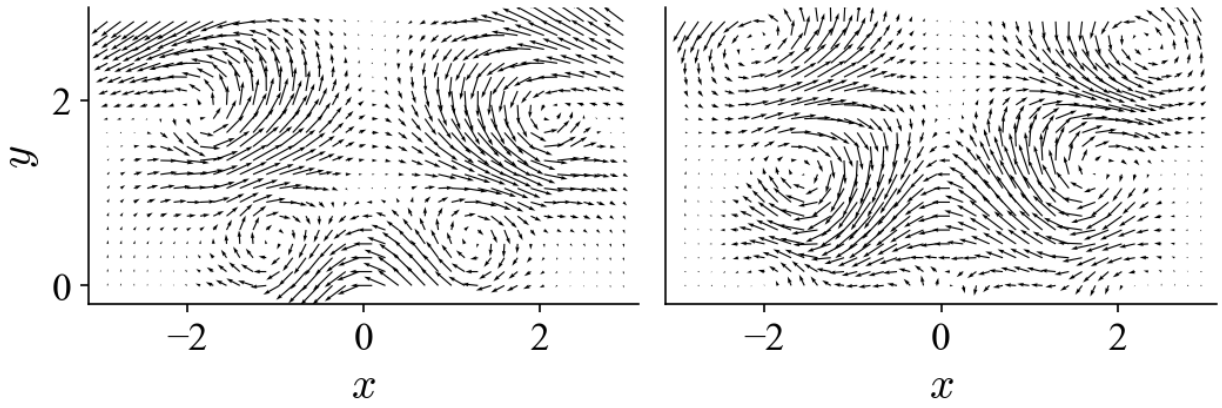
Note: The same dataset can be created using functions given in `examples.vortex_shedding`.

POD of the dataset

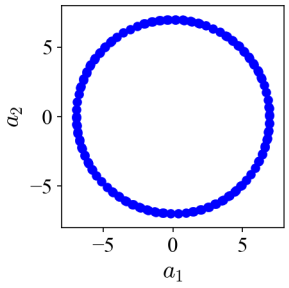
Using the built-in function `pod_modes`, the process can be carried out on the dataset `v_array` (containing 400 frames at 10 kHz sampling rate, the vortex dynamics is set at 470 Hz) as:

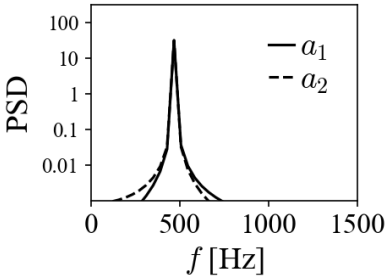
```
from mrpod import pod_modes

# v_array is the pre-generated dataset
pod_results = pod_modes(v_array, num_of_modes=4, normalize_mode=True)

# get the modes and projection coefficients
proj_coeffs = pod_results['proj_coeffs']
modes = pod_results['modes']
eigvals = pod_results['eigvals']
# normalize eigenvalues
eigvals = eigvals/eigvals.sum()*100
```

Normalizing the eigenvalues is a common practice to get a sense of the contribution of the POD modes to the total kinetic energy. From the results we can see that the first two POD modes have nearly identical eigenvalues and compose nearly 100% of the kinetic energy. If we visualize the two POD modes in the same fashion as the vector fields above, we get:



As can be seen, these two POD modes look nearly identical to the two modes used to construct the dataset. The results suggest that two POD modes are needed to describe a traveling vortex in the flow field. Bearing this in mind, let's look at the projection coefficients of these two modes:





When the projection coefficients are plotted against each other in a so-called phase portrait, they fall onto a perfect circle, indicating the phase shift of 90 degrees. A peak at 470 Hz can be identified in both of their power spectra densities (PSD). Both the phase shift and the peak frequency match the values used to generate the traveling vortices in the first place.

Without prior knowledge of how the flow dynamic is created, it is perhaps not immediately clear what we should make of the POD modes. The example shown here aims to answer this question: how can we identify coherent structures in the flow field (or similar environments) from the POD results? The clues can be found above and can be summarized below:

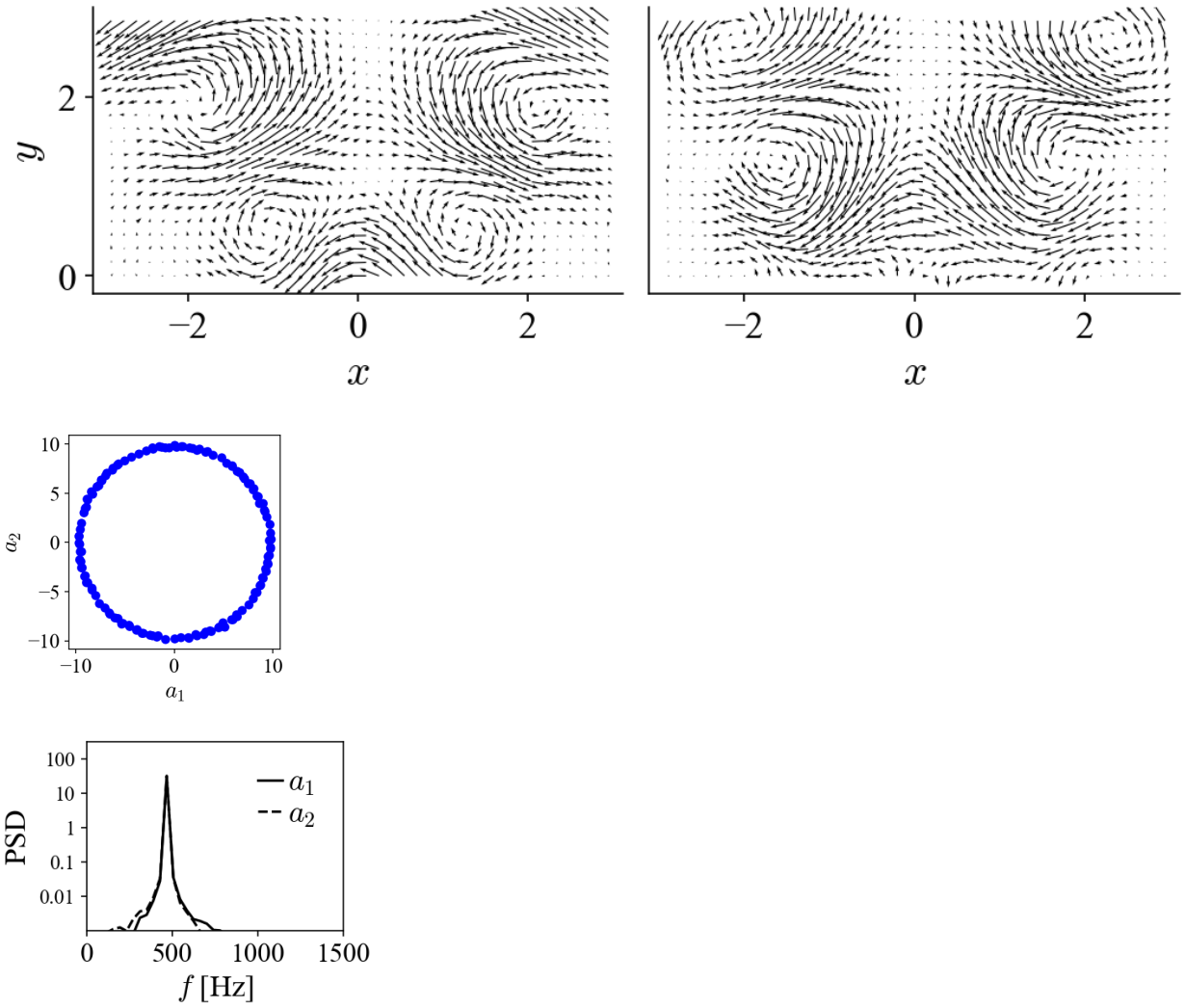
- Two POD modes are necessary to describe a traveling structure;
- They should have similar spatial appearance and comparable eigenvalues;
- Their projection coefficients should exhibit a regular correlation in the phase portrait,
- which should have very similar footprints in the spectral domain.

These 4 criteria should be considered when trying to recognize physical flow patterns based on data-driven POD.

Reduced-order reconstruction

Sometimes it is not immediately clear from the POD modes what flow pattern they represent. It is therefore useful to visualize the flow pattern, especially in the case of noisy dataset, such as the following:

This dataset is identical to the one shown on the top of this page but with added random (white) noise in each frame to obscure the pattern of the traveling vortices. From POD of the dataset (also 400 frames at 10 kHz), we get (first two modes):



The results are nearly identical to the ones from the original dataset. It is clear that with this noise level POD has no problem of extracting the modes associated to the flow pattern. Since now that the flow pattern is not immediately clear from the noisy dataset, can we somehow visualize it with the POD modes? Recall how the original dataset is generated and analogously we can "reconstruct" the dataset with selected modes according to

$$\boldsymbol{v}_{\text{reduced}} = \sum_{i=1}^{n} a_i \boldsymbol{\Phi}_i$$

where $n \leqslant N$ (N is the total number of modes with non-zero eigenvalues). If we include just the two modes corresponding to the traveling vortices, the equation becomes essentially equivalent to the one shown on the top and the "reduced-order" flow field becomes:

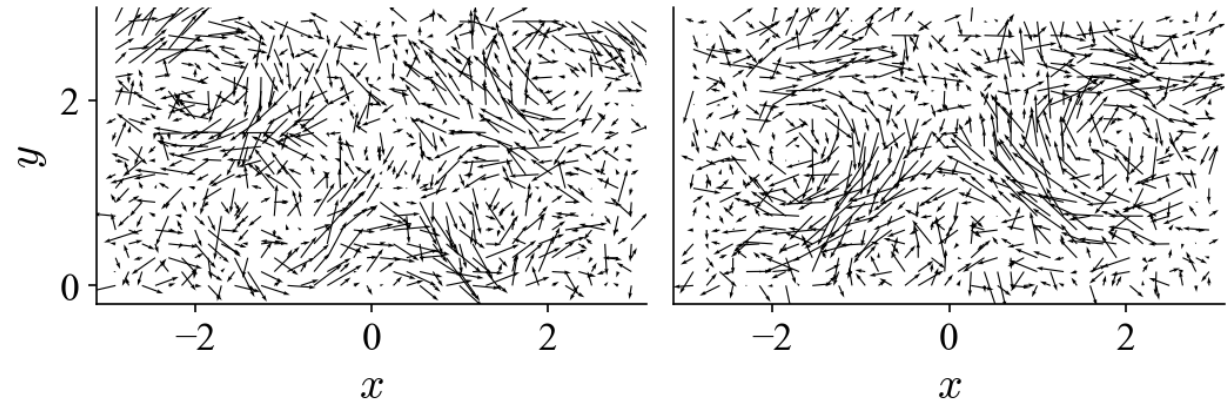
So now we have a visual idea what the POD modes entail. This also shows how POD can be used to denoise a dataset, i.e., by leaving out noisy modes during the reduced-order reconstruction.

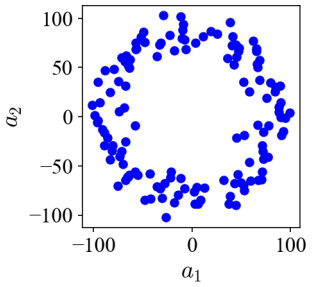
Where POD fails

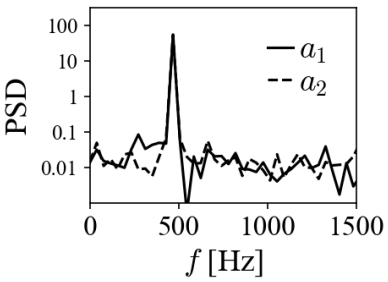
Sub-noise-level dynamics

We have seen how POD can be used to denoise a dataset and extract obscured flow pattern from it. There is however a limit. When the flow pattern is overwhelmed by noise (in terms of kinetic energy), POD won't perform as well, as shown for the noisier dataset below:

The noise level has been cranked way up. The POD results below are quite noisy to the point that they cannot really be used to unambiguously visualize the hidden flow pattern (only the first two modes are shown):







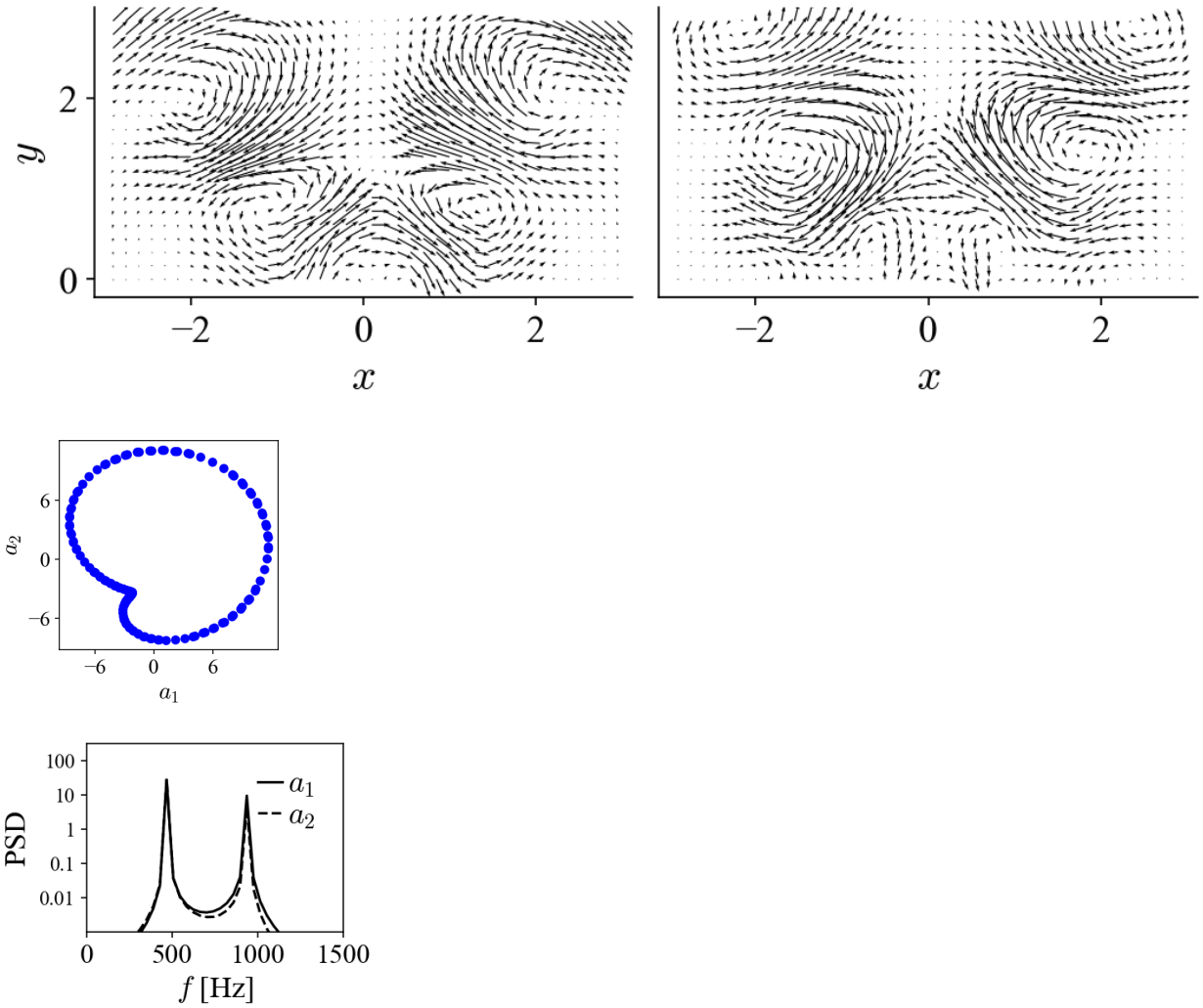
Coexistence of multiple dynamics

Another drawback of POD is that it is a purely energy-based decomposition process and it disregards all temporal correlations in the dataset. Even if we were to randomly shuffle the 400 frames in the datasets above, we would get exactly the same results (we wouldn't be able to get the frequency of the traveling vortices though). This lack of so-called "dynamic ranking" becomes quite problematic in a scenario where multiple dynamics coexist across a wide range of time scales.

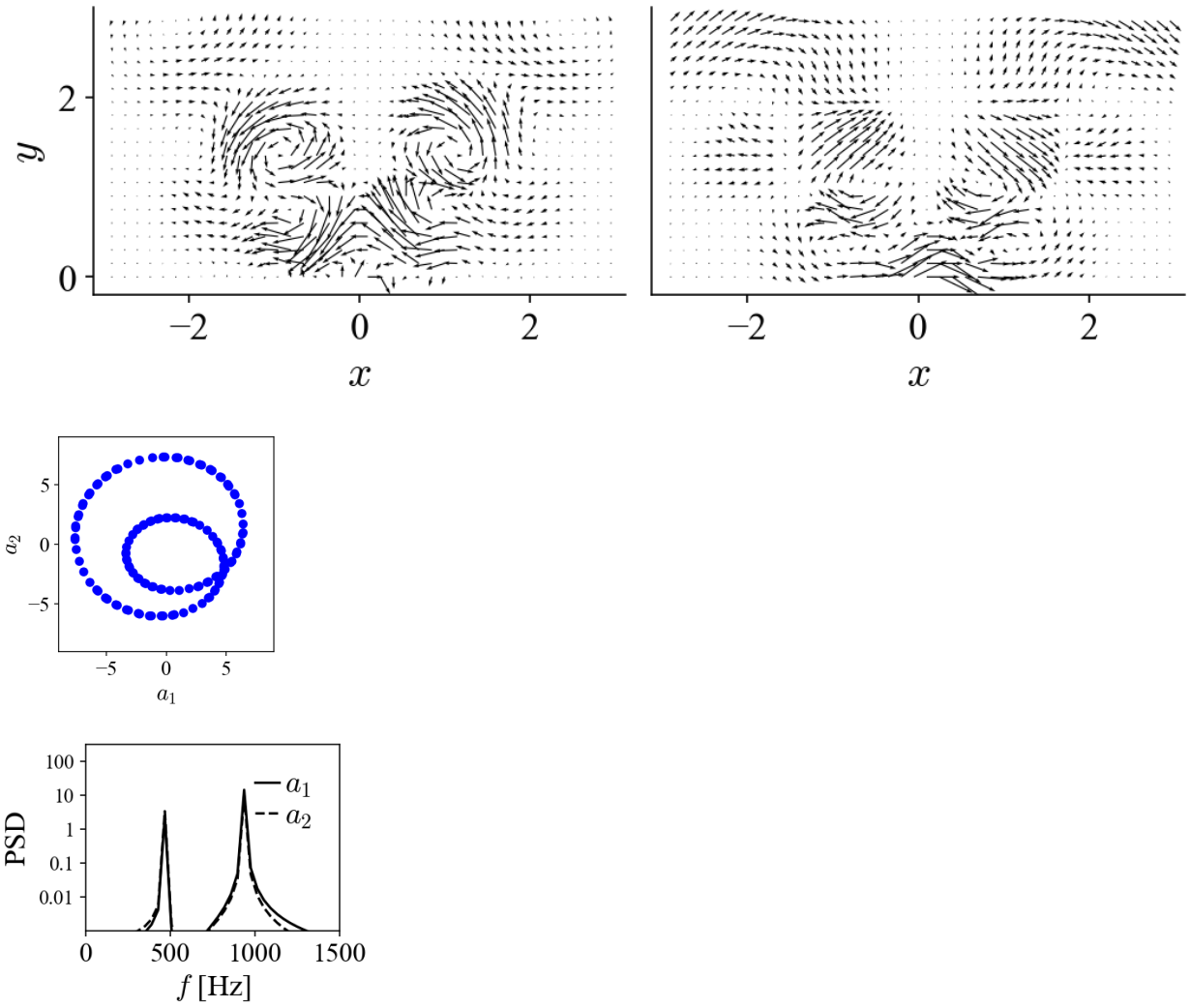
To demonstrate this, we can introduce another vortex pattern into the dataset that has different spatial and temporal behaviors from the one above:

And our goal now is try to decompose the new mixed dataset below to separate these two flow patterns:

If we perform POD on this dataset, we get the first two modes (mode 1 and 2):



and the following two modes (mode 3 and 4):



It is obvious that POD does not just automatically "group" or "isolate" the same dynamic into two modes. Instead, it essentially lumps different dynamics and distribute them among several modes (four modes in this case). Neither the spatial modes nor their projection coefficients possess the spectral purity to allow unambiguous interpretation of the underlying dynamics.

Warning: From these two examples it is clear that POD modes do not equate physical patterns. It is always necessary to first understand the underlying physics (in this case, the traveling structures) before attempting to interpret the POD results.

See also:

To fix this issue, we need to introduce dynamic ranking into the POD process. In the next tutorial *Pattern recognition with MRPOD*, MRPOD is demonstrated on these two "challenging" datasets to showcase its capabilities.

1.4.2 Pattern recognition with MRPOD

Note: Please refer to the previous tutorial *Pattern recognition with POD* for more details regarding the synthesized datasets and the performances of POD in pattern recognition. Instead of `pod_modes`, the function `mrpod_detail_bundle` will be used to carry out the task.

We will pick up right where we left in the previous tutorial and use the same examples to demonstrate the advantages of MRPOD over POD for pattern recognition in the flow field.

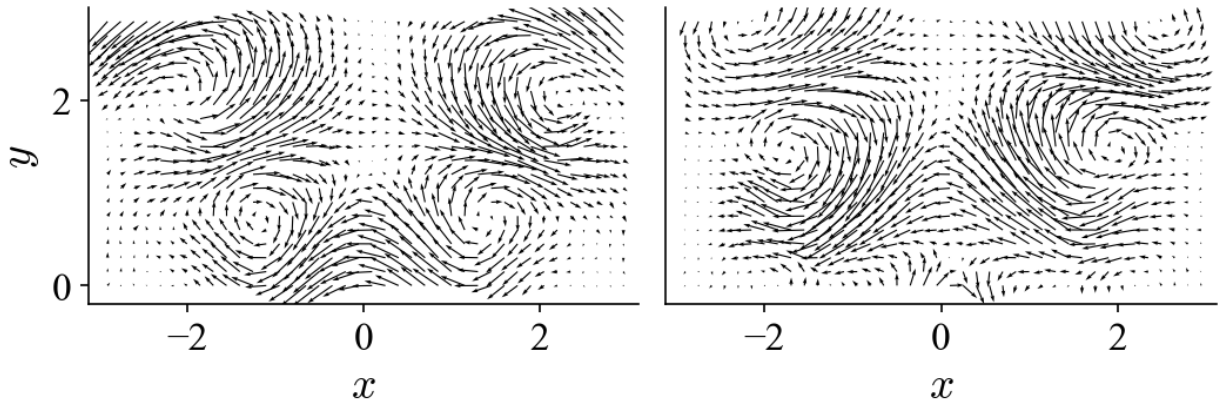
Sub-noise-level dynamics

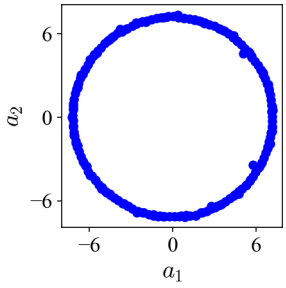
For the very noisy dataset created to challenge POD:

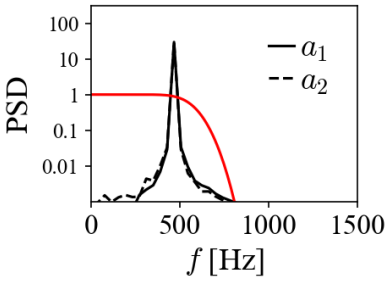
By performing MRPOD within a shortpass imposed by the composite wavelet filter,

```
from mrpod import mrpod_detail_bundle

# pre-generated filterbank with symlet of the length 24
dir_filterbank = "filters_sym12_j=8.pkl"

# set decomposition level j and scale n
j_level = 3
n_scale = 0

# v_array is the pre-generated dataset
pod_results = mrpod_detail_bundle(v_array, js=[j_level], scales=[n_scale],
                                  seg=1, num_of_modes=10,
                                  full_path_filterbank=dir_filterbank)
# get the modes and projection coefficients
proj_coeffs = pod_results['proj_coeffs']
modes = pod_results['modes']
eigvals = pod_results['eigvals']
# normalize eigenvalues
eigvals = eigvals/eigvals.sum()*100
```

we can obtain the following two modes (mode 1 and 2):





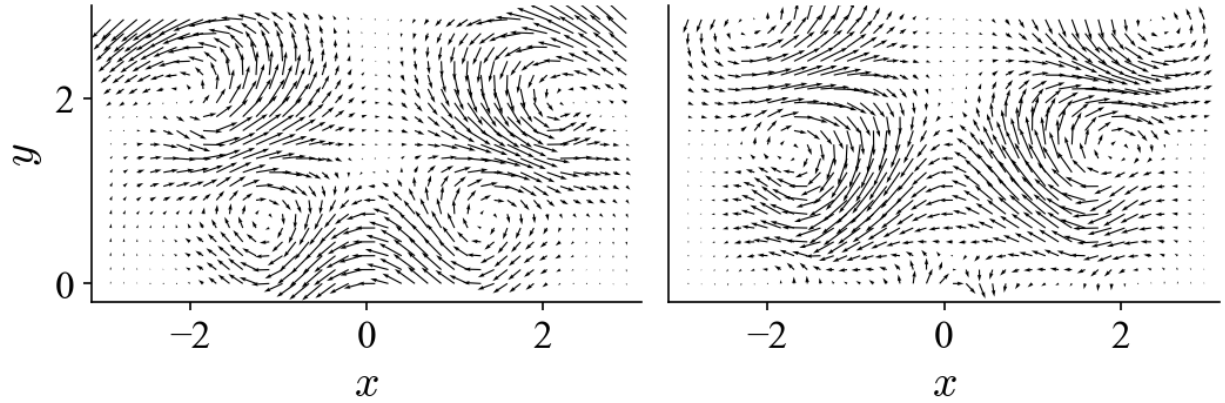


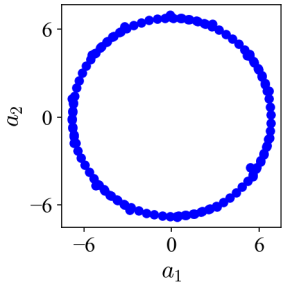
Comparing to the results achieved with POD in the previous tutorial, the superior performance of MRPOD on this problem is quite striking. The bandpass (shown as the red line in the phase portrait) can be narrowed to further improve the denoising capability.

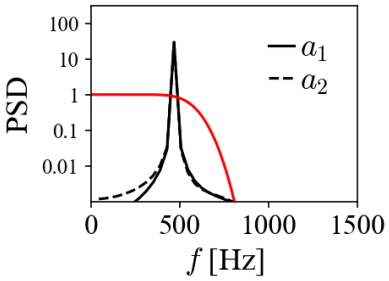
Coexistence of multiple dynamics

For the dataset with two superpositioned dynamics:

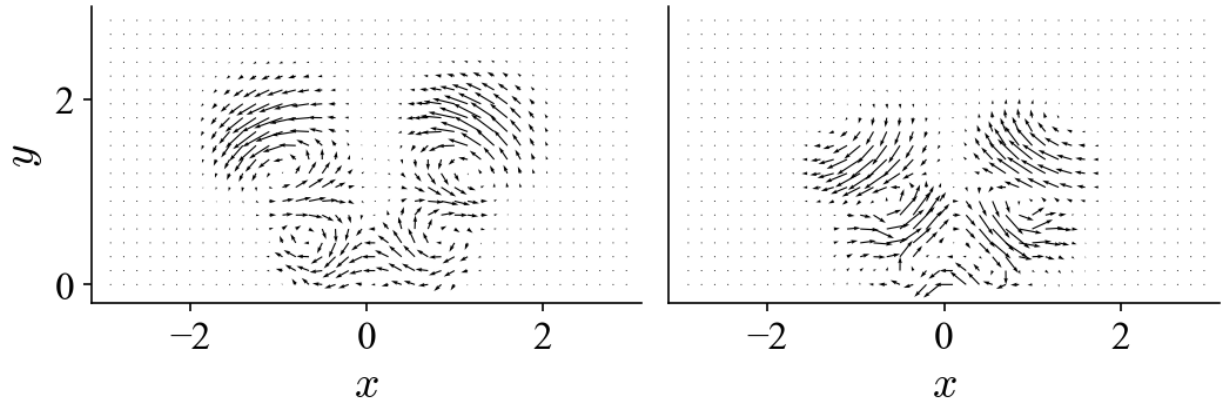
We can design two bandpasses to isolate the two distinct dynamics in the spectral domain and carry out MRPOD accordingly. For the first (original) dynamic we can impose a lowpass filter and get:

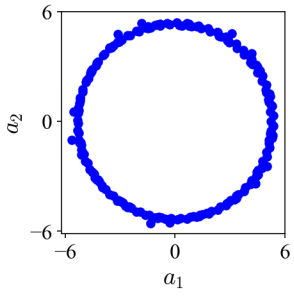


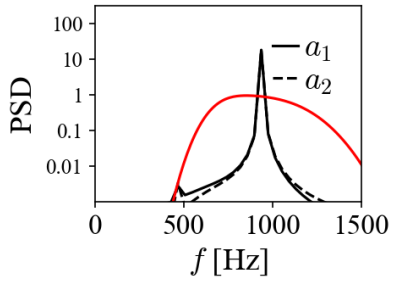




Analogously we can impose another bandpass filter to extract the added dynamic (by setting *n_scale=1* in the Python script above):







Now we have separated these two dynamics and we can inspect them without the spectral cross-talk that we saw in the previous tutorial with POD. Using the reduced-order reconstruction introduced in the previous section, we can visualize these two dynamics separately as:

Warning: The composite wavelet filters must be tailored to the specific problem at hand by considering the necessary spectral isolations, the length of the dataset and the desired outcome. MRPOD is *not* a one-size-fits-all technique.

Bibliography

[MRDMD] Kutz, J., Fu, X., Brunton, S. Multiresolution dynamic mode decomposition. *SIAM Journal on Applied Dynamical Systems* 15 (2), 713-735, 2016.

[mPOD] Mendez, M. A., Balabane, M., Buchlin, J. M. Multi-scale proper orthogonal decomposition of complex fluid flows. *Journal of Fluid Mechanics* 870, 988-1036, 2019.

[MRPOD] Yin, Z., Stöhr, M. Time–Frequency Localisation of Intermittent Dynamics in a Bistable Turbulent Swirl Flame. *Journal of Fluid Mechanics* 882, A30, 2020.

Python Module Index

m

Index